```
//-----------------------------------------------------------------------------------------------------------------
// MPI - MESSAGE PASSING INTERFACE
//-----------------------------------------------------------------------------------------------------------------
        #include<iostream>
        #include<mpi.h>

        int main(int argc, char *argv[]){
            MPI_Init(&argc, &argv);
            int rank, size;
            MPI_Comm_size(MPI_COMM_WORLD, &size);      // size = #processors
            MPI_Comm_rank(MPI_COMM_WORLD, &rank);      // rank = ID of the processor
            .. ;
            MPI_Finalize();
            return 0;
                }
// * if we pass 0 arguments -> argc=1, argv[0] is the name of the program
// * if we pass k arguments -> argc=k+1, argv[1] first parameter, .., argv[k] k-th input

// COMPILE and RUN
        $ mpicxx --std=c++11 file_name.cc -o exe_name                  // 1.
        $ mpicxx --std=c++11 file_name.cc                              // 2.
        $ mpicxx --std=c++11 file_name.cc file1.cc file2.cc -o exe_name // 3. (e.g. with classes)
        $ mpicxx --std=c++11 file_name.cc file1.cc file2.cc            // 4.

        $ mpiexec -np=4 exe_name                                       // 1.a, 3.a
        $ mpiexec -np=4 exe_name text1 text2                           // 1.b, 3.b
        $ mpiexec -np=4 a.out                                          // 2.a, 4.a
        $ mpiexec -np=4 a.out text1 text2                              // 2.b, 4.b
// * 1.a, 2.a, 3.a, 4.a: running on 4 processors, argc=1, argv[0] is the name of the program
// * 1.b, 2.b, 3.b, 4.b: running on 4 processors, argc=3, argv[1]="text1", argv[2]="text3

//-----------------------------------------
// COMMUNICATORS
//-----------------------------------------
// * point-to-point: involves two processors -> sender, receiver
// * collective    : involves all processors -> broadcast, reduce, scatter, gather,

// ********************
// SEND and RECEIVE
// ********************

        MPI_Send(const void *buf,                   // where is what we want to send?
                 int count, MPI_Datatype datatype,  // how many (at most) and of what type?
                 int dest, int tag, MPI_Comm comm)  // to who? tag? through what?

        MPI_Recv(void *buf,                          // where do we store?
                 int count, MPI_Datatype datatype,   // how many (at most) and of what type?
                 int source, int tag, MPI_Comm comm, // from who? tag? through what?
                 MPI_Status *status)                 // "MPI_STATUS_IGNORE"

// Example: rank0 takes as input a double and sends it to the others
        double n;
        if(rank==0){
            cin >> n;
            for(int dest=1; dest<size; ++dest)
                MPI_Send(&n, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
        }
        else{
            MPI_Recv(&n, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

// ********************
// BROADCAST
// ********************
// Rank_0 sends a message to all the other ranks, without doing it one by one:
// * cannot use tags
// * every other process in the communicator MUST call the collective function

        MPI_Bcast(void *buffer,                     // sender: where is stored the info
                                                    // receiver: where to store the info
                  int count, MPI_Datatype datatype, // how many (at most) and of what type?
                  int root, MPI_Comm comm)          // who owns the data? source. Through what?

// Example: rank0 takes as input a double and sends it to the others
        double n;
        if(rank==0)
            cin >> n;
        MPI_Bcast(&n, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

// ********************
// REDUCE/ALL_REDUCE
// ********************
// Perform efficiently some operations (max, min, sum, prod) among ranks.

        MPI_Reduce(const void *sendbuf, void *recvbuf,  // send buffer, receiver buffer
                   int count, MPI_Datatype datatype,    // how many (at most) and of what type?
                   MPI_Op op, int dest, MPI_Comm comm)  // what operation? send to who? through what?

// Example: every rank has a local value and we want the sum of them in rank_0
// Note: we decleare "total" in each rank, not only in the destination one.
        double local_value;
        double total;
        MPI_Reduce(&local_value, &total, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

```
// What if we want the sum to be saved in the variable total of each rank?

        MPI_Allreduce(const void *sendbuf, void *recvbuf,  // send buffer, receiver buffer
                      int count, MPI_Datatype datatype,    // how many (at most) and of what type?
                      MPI_Op op, MPI_Comm comm)            // what operation? through what?

// MPI_IN_PLACE
// With MPI_IN_PLACE we can use a single buffer for both input and output.
// (we cannot do, for example: MPI_Allreduce(&minimum, &minimum, ... ); )
        double minimum;
        MPI_Allreduce(MPI_IN_PLACE, &minimum, 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);

// ********************
// SCATTER and GATHER
// ********************
// When we have to divide data between ranks we can procede in two ways:
// CYCLIC PARTITIONING: used when data source is available across all processes

                for(size_t i=rank; i<v.size(); i+=size) { .. }

// BLOCK PARTITIONING: used when data source is available on a single process (#elems % #process = 0)

        MPI_Scatter(const void *sendbuf,                  // send buffer
                    int sendcount, MPI_Datatype sendtype, // #elems we send to individual rank? type?
                    void *recvbuf,                        // receiver buffer
                    int recvcount, MPI_Datatype recvtype, // #elems will be received? type?
                    int root, MPI_Comm comm)              // source of the data? through what?

// The dual of MPI_Scatter (decomposes one into many) is MPI_Gather (composes many into one):

        MPI_Gather(const void *sendbuf,                   // send buffer
                   int sendcount, MPI_Datatype sendtype,  // #elems we send? type?
                   void *recvbuf,                         // receiver buffer
                   int recvcount, MPI_Datatype recvtype,  // #elems will be received? type?
                   int root, MPI_Comm comm)               // destination of the data? through what?

// If we want the send buffer to be update in all ranks, then:

        MPI_Allgather(const void *sendbuf,                  // send buffer
                      int sendcount, MPI_Datatype sendtype, // #elems we send? type?
                      void *recvbuf,                        // receiver buffer
                      int recvcount, MPI_Datatype recvtype, // #elems will be received? type?
                      MPI_Comm comm)                        // through what?

//-------------------------------------------
// READ_VECTOR and PRINT_VECTOR
//-------------------------------------------

        vector<double> x = mpi::read_vector(n, "x", MPI_COMM_WORLD);
        vector<double> y = mpi::read_vector(n, "y", MPI_COMM_WORLD);
        vector<double> z = x+y;
        mpi::print_vector(z, n, "The result is ..", MPI_COMM_WORLD);

// mpi::read_vector : read and spread the informations among processes (MPI_Scatter)
// mpi::print_vector: build a single vector from locals and print it (MPI_Gather, rank_0)

//-------------------------------------------
// Random REMEMBER
//-------------------------------------------
// 1. When inside the code we have that rank_0 takes something in input (e.g. cin>>a>>b>>c;) we do:
        $ mpicxx --std=c++11 file_name.cc
        $ mpiexec -np 4 a.out
        5 6 7
//    In this way we perform the assignment: a=5, b=6, c=7. More fast:
        $ vim input_short                                  // creates a file where we write "5 6 7"
        $ cat input_short                                  // opens the file
        5 6 7
        $ mpiexec -np 4 a.out < input_short                // the "<" means "take inputs from .. "
        $ mpiexec -np 4 a.out < input_short > output_short // output saved in "output_short" (nothing displayed)

// 2. * STANDARD INPUT: if we have std::cin (rank_0) in the code -> separate line of inputs in the cmd line
//      (after this rank_0 has to pass the informations through broadcast/scatter/send)
        $ mpicxx --std=c++11 file_name.cc
        $ mpiexec -np 4 a.out
        5 6 7                                      // cin>>a>>b>>c

//    * USER PASSES INFORMATIONS: if we get something from the arguments -> we write arguments in the cmd line
//      (after this the information is available for all the processes)
        $ mpicxx --std=c++11 file_name.cc
        $ mpiexec -np 4 a.out 5 6 7
//      ALL processors have the informations, but we need conversions (argv is a vector of strings).
        if(argc == 4){                             // we have arguments -> argc = 1 + #arguments
            unsigned var1 = std::stoul(argv[1]);   // var1 = 5; (unsigned)
            unsigned var2 = std::stoul(argv[2]);   // var2 = 6; (unsigned)
            int      var3 = std::stoi(argv[3]);    // var3 = 7; (integer)
        }

// 3. We can use MPI_Reduce/MPI_Allreduce also on vectors
        int N = 10;
        vector<double> local_x(N), sum(N);
        MPI_Reduce(local_x.data(), sum.data(), N, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

for vectors and matrices

MPI_Bcast(&the_string[0], length, MPI_CHAR, 0, MPI_COMM_WORLD);

```cpp
#include<iostream>
#include<mpi.h>

int main(int argc, char* argv[]){

        MPI_Init(&argc, &argv);

        int rank, size;
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        .. ;

        MPI_Finalize();
        return 0;
}

// Send/Receive (point-to-point)
MPI_Send(&n, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
MPI_Recv(&n, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

// Broadcast (collective)
MPI_Bcast(&n, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

// Reduce/Allreduce (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD)
MPI_Reduce(&local_value, &total, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Allreduce(MPI_IN_PLACE, &value, 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);

// Cyclic partitioning
for(size_t i=rank; i<v.size(); i+=size){ .. }

// Block partitioning: Scatter/Gather
MPI_Scatter(global.data(), local_n, MPI_DOUBLE, local.data(), local_n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Gather(local.data(), local_n, MPI_DOUBLE, global.data(), local_n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_AllGather(local.data(), local_n, MPI_DOUBLE, global.data(), local_n, MPI_DOUBLE, MPI_COMM_WORLD);

// Read_vector/Print_vector
vector<double> x = mpi::read_vector(n, "x", MPI_COMM_WORLD);
vector<double> y = mpi::read_vector(n, "y", MPI_COMM_WORLD);
vector<double> z = x+y;
mpi::print_vector(z, n, "The global vector is: ", MPI_COMM_WORLD);


//-------------------------------------------------------------------------------------------------------------------------

$ mpicxx --std=c++11 file_name.cc
$ mpiexec -np=4 a.out

// Name specification
$ mpicxx --std=c++11 file_name.cc -o exe_name
$ mpiexec -np=4 exe_name

// More files
$ mpicxx --std=c++11 file_name.cc file1.cc file2.cc -o exe_name
$ mpiexec -np=4 exe_name

// Arguments
$ mpicxx --std=c++11 file_name.cc -o exe_name
$ mpiexec -np=4 exe_name text1 text2              // argc=3, argv[1]="text1", argv[2]="text2"

// Standard input (only rank_0 has "std::cin >> a >> b >> c;")
$ mpicxx --std=c++11 file_name.cc -o exe_name
$ mpiexec -np=4 exe_name
5 6 7                                             // a=5, b=6, c=7

// Standard input with saved input and output
$ vim input_short
$ mpicxx --std=c++11 file_name.cc -o exe_name
$ mpiexec -np=4 exe_name < input_short > output_short

// User passed informations
$ mpicxx --std=c++11 filne_name.cc -o exe_name
$ mpiexec -np=4 exe_name 5 6 7
// ..
if (argc==4){
        unsigned var1 = std::stoul(argv[1]);
        unsigned var2 = std::stoul(argv[2]);
        int      var3 = std::stoi(argv[3]);
}
```